Gram schmidt Factorisation

## FUNCTION : GRAM-SCHMIDT ORTHOGONALIZATION

**Purpose:** This function implements the Gram-Schmidt orthogonalization process, which transforms a set of linearly independent vectors (columns of matrix A) into an orthonormal basis (stored in matrix Q).

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation:**

1. **First Vector Initialization**:
   * The function copies the first column of matrix A directly to matrix Q.
   * Q[i] = A[i]; for all rows i from 0 to n-1.
2. **Orthogonalization Process**:
   * The outer loop for (int i = 1; i < n; i++) iterates through each column of A starting from the second column.
   * For each column i:
     + Initially copies column i from A to Q: Q[j][i] = A[j][i];
     + For each previously processed column k (0 to i-1):
       - Calculates the dot product between column k of Q and column i of A: vector\_product+=Q[l][k]\*A[l][i];
       - Calculates the squared norm of column k of Q: norm += Q[l][k] \* Q[l][k];
       - Subtracts the projection of column i onto column k: Q[j][i] -= (vector\_product\*Q[j][k]) / norm;
     + This makes column i orthogonal to all previous columns.
3. **Normalization**:
   * After all columns are orthogonalized, each column is normalized to have unit length.
   * For each column i:
     + Calculates the Euclidean norm: norm = sqrt(norm);
     + Divides each element by the norm: Q[k][i] /= norm;
     + This transforms orthogonal vectors into orthonormal vectors.
4. **Result Output**:
   * The function prints the resulting orthonormal matrix Q.
   * Each row is displayed in bracket notation with values formatted to 5 decimal places.

## **Function: Backward Substitution for Solving Upper Triangular Systems**

**Purpose:** This function solves a system of linear equations **Ux = b**, where **U** is an upper triangular matrix.

A computer screen shot of a program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * double \*x = (double \*)malloc(n \* sizeof(double));  
     Allocates memory for the solution vector **x**.
   * Checks for allocation failure and exits if memory is unavailable.
2. **Initialize Last Element**:
   * x[n-1] = b[n-1] / A[n-1][n-1];  
     Solves for the last variable directly using the bottom row of the upper triangular matrix **U**.
3. **Backward Substitution Loop**:
   * for(int i=n-2; i>=0; i--)  
     Iterates from the second-last row to the first row.
   * double sum = 0;  
     Initializes a sum to accumulate contributions from already solved variables.
   * for(int j=i+1; j<n; j++)  
     Sums the products of coefficients (A[i][j]) and known solutions (x[j]).
   * x[i] = (b[i] - sum) / A[i][i];  
     Computes the current variable x[i] by subtracting the accumulated sum from b[i] and dividing by the diagonal element A[i][i].
4. **Return Result**:
   * Returns the solution vector **x**, which satisfies **Ux = b**.

## FUNction : Matrix Transposition

**Purpose:** This function transposes a **square matrix** of size n x n

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Outer Loop** (i from 0 to n-1):
   * Iterates over each row of the matrix.
2. **Inner Loop** (j from i+1 to n-1):
   * For each row i, iterates over columns starting from i+1 (avoids redundant swaps on the diagonal and below).
3. **Swap Elements**:
   * Swaps A[i][j] (element above the diagonal) with A[j][i] (element below the diagonal).
   * Example: For i=0 and j=1, A and A are swapped.
4. **Result**:
   * After all iterations, the original matrix A becomes its transpose.

## Function : Matrix Multiplication for Square Matrices

**Purpose:** This function multiplies two **square matrices** A and B of size n x n and stores the result in matrix C.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * The outer loop for (int i = 0; i < n; i++) iterates over each **row** of matrix A.
2. **Column Iteration**:
   * The middle loop for (int j = 0; j < n; j++) iterates over each **column** of matrix B.
3. **Dot Product Calculation**:
   * C[i][j] = 0; initializes the element at row i, column j of matrix C to zero.
   * The inner loop for (int k = 0; k < n; k++) computes the dot product of the i-th row of A and the j-th column of B:
     + C[i][j] += A[i][k] \* B[k][j]; accumulates the product of corresponding elements.
4. **Result Storage**:
   * After completing all iterations, matrix C contains the product of A and B.

## Function : Matrix-Vector Multiplication

**Purpose:** This function multiplies a **square matrix** A of size n x n with a **vector** B of size n and stores the result in vector C. It computes the dot product of each row of A with B to produce C.

A computer screen shot of a black screen with colorful text

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Initialization**:
   * The outer loop for (int i = 0; i < n; i++) iterates over each **row** of matrix A.
2. **Reset Result Vector**:
   * C[i] = 0; initializes the i-th element of the result vector C to zero.
3. **Dot Product Calculation**:
   * The inner loop for (int j = 0; j < n; j++) computes the dot product of the i-th row of A and the vector B:
     + C[i] += A[i][j] \* B[j]; accumulates the product of the matrix element A[i][j] and vector element B[j].
4. **Result Storage**:
   * After the inner loop completes, C[i] holds the dot product result for the i-th row.

## Function : QR Factorization for Solving Linear Systems

**Purpose:** This function solves a linear system **Ax = b** using QR factorization. It decomposes matrix **A** into **Q** (orthonormal) and **R** (upper triangular), then solves **Rx = Qᵀb** via backward substitution.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Memory Allocation**:
   * Allocates memory for matrices **Q** (orthonormal) and **R** (upper triangular).
2. **Gram-Schmidt Process**:
   * Gram\_schmidt(n, A, Q) transforms **A** into **Q** with orthonormal columns.
3. **Transpose Q**:
   * transpose(n, Q) converts **Q** from column-orthonormal to row-orthonormal (effectively computing **Qᵀ**).
4. **Compute R**:
   * matrix\_multiply\_nn(n, Q, A, R) calculates **R = Qᵀ \* A**.
5. **Transform b**:
   * matrix\_multiply\_n1(n, Q, b, b1) computes **b1 = Qᵀ \* b**.
6. **Solve for x**:
   * backward\_substitution(n, R, b1) solves **Rx = b1** to find the solution vector **x**.
7. **Memory Cleanup**:
   * Frees **Q**, **R**, and **b1** to prevent leaks.

## function : Reading Matrix and Vector Input from a File

**Purpose:** This function reads a square matrix **A** of size n x n and a vector **b** of size n from a file named "inputs.txt". It ensures proper error handling for file operations and input parsing.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **Open File**:
   * FILE \*file = fopen("inputs.txt", "r"); opens the file "inputs.txt" in read mode.
   * If the file cannot be opened (e.g., it doesn't exist), the program prints an error message ("File not found") and exits.
2. **Read Matrix Elements**:
   * The nested loop for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) { ... } } iterates through each row (i) and column (j) of matrix **A**.
   * fscanf(file, "%lf", &A[i][j]) reads a floating-point number from the file and stores it in the corresponding position in **A**.
   * If the input is invalid or missing, the program prints "Invalid input" and exits.
3. **Read Vector Elements**:
   * The loop for (int i = 0; i < n; i++) { ... } iterates through each element of vector **b**.
   * fscanf(file, "%lf", &b[i]) reads a floating-point number from the file and stores it in the corresponding position in **b**.
   * Similar error handling ensures that invalid or missing input causes the program to terminate with an error message.
4. **Close File**:
   * fclose(file); closes the file after all data has been read to release system resources.

## FUNCTION: Main Program for Solving Linear Systems via QR Factorization

**Purpose:** This program solves a linear system **Ax = b** using QR factorization. It reads matrix **A** and vector **b** from a file, computes the solution **x**, prints the result, and measures the execution time of the QR factorization process.

A screen shot of a computer program

AI-generated content may be incorrect.

**Step-by-Step Explanation**

1. **User Input**:
   * Prompts the user to enter the number of variables (n).
2. **Memory Allocation**:
   * Allocates memory for:
     + **A** (n x n matrix) using nested malloc calls.
     + **b** (vector of size n).
   * Checks for allocation failure in b and exits if memory is unavailable.
3. **Read Input File**:
   * Calls read\_file\_input(n, A, b) to populate **A** and **b** from "inputs.txt".
4. **QR Factorization & Timing**:
   * clock() records the start time.
   * QR\_factorization(n, A, b) computes the solution vector **x**.
   * clock() records the end time.
5. **Print Results**:
   * Displays the solution vector **x** with 6 decimal places.
   * Calculates and prints the execution time in nanoseconds.
6. **Memory Cleanup**:
   * Frees all dynamically allocated memory for **A**, **b**, and **x** to prevent leaks.
7. Key note:
   * The clock() function used to measure execution time in this program works reliably when running in **Windows Subsystem for Linux (WSL)** via **VS Code**. However, if you run the same code in a terminal-based VS Code setup (outside WSL), the behaviour of clock() may vary or provide poor precision due to platform-specific differences in how the function is implemented.
   * To ensure accurate timing, it is recommended to use this program within a WSL-based VS Code environment.